Logo

Description automatically generatedProject Report– Formal Languages and Automata Theory

A person in a garment

Description automatically generated with low confidence

# Khizar Ali Shah (2020196)

# Muhammad Omer (2020335)

Implementation of an algorithm for creating a deterministic finite state machine from a regular expression

# Introduction

The goal of this project is to develop an algorithm for conversion of regular expression to a finite state machine. From Kleene’s theorem we know that every finite automata is a transition graph, and that every transition graph has a regular expression. Therefore, the third part of the theorem concludes that every regular expression has an associated finite automata. Kleene’s Theorem shows us the equivalence of regular expression (RE) and finite state machines or finite automata (DFA).

The proof of this part gives us an insight as to how the algorithm for this purpose can be developed. However, it only gives us the understanding and the pseudocode, not the actual implementation in any programming language. Hence in this project, we implemented this algorithm using Data Structures and Object-oriented programming paradigm of the C++ programming language.

# Working

# Separating the interface from implementation

The key idea of object-oriented programming is encapsulation. This can be achieved by separating the function and class declarations from their definition or interface. We applied this concept, and the resulting files are as follows



*state.h* and *fa.h* files contain the declarations of the classes *states* and *finite automata.* The definitions of the member functions of these classes are in *state.cpp* and *fa.cpp* respectively. Applying the inclusion rule of C++, the *.h* files are included in the *.cpp* files so that the linker can link the function declarations with their implementation during the linking process after the compilation of individual *.cpp* files is done by the compiler and a *.o* file is created for each. The linker will create an executive file (*.exe)* which will be loaded by the loader on the main memory and run.

# State class

Before moving on to the declaration of state class, it is must that we discuss how a state object, or a state looks like and what are its characteristics. A state class object can be visually looked as follows

name readA\*

isFinal readB\*

The declaration of *state.h* is as follows

Text

Description automatically generated

Fig. 1

The state object can be created dynamically as follows

Graphical user interface

Description automatically generated with low confidence

The *new* keyword will allocate memory for this object in the *heap* section of the program. After this, it will call the constructor which is declared in Fig.1 line 15 and defined as follows in the *state.cpp*

Text

Description automatically generated

*count* is the static member of the class. It is associated with the class, and not any object, and is initialized to 1 initially. Every time a state is created, its name will be set as 1, 2, 3, and so on. To get rid of garbage values present in the member variables of the object, they are initialized to *nullptr* and *false.* By default, a state created is not a final state, and will be set as a final state, if need be, in another function discussed later.

In an *FA*, the next state from a given state can be any state reached by reading *a* or *b*. It may or may not be the next state in order of its name, i.e., 1, 2, 3, and so on. We need to keep track of this order so that the printing of *transition table* of states can be done properly. For this purpose, a state has a *nextStateInOrder* pointer which points to the next state according to the name, as the name of this pointer suggests.

The *operator=* is the operating overloading function for the *assignment* operator, which will be used for deep copy of an object into another. It will be required later.

# FA Class

A ***finite automaton*** is a 5-tuple (Q, Σ, δ, q0, F ), where

**1.** Q is a finite set called the ***states***,  
**2.** Σ is a finite set called the ***alphabet***,  
**3.** δ : Q × Σ−→ Q is the ***transition function***,  
**4.** q0 ∈ Q is the ***start state***, and  
**5.** F ⊆ Q is the ***set of accept states***.

The declaration of this class is as follows

Text

Description automatically generated

Fig. 2

The start state’s pointer keeps the *linked list* of all the states in the memory. This way, point 1 and 4 of the definition is fulfilled. Transition functions will be developed in the *createFiniteAutomata* function discussed in the next heading. As mentioned earlier, the final state information will be kept by the FA.

# Creating Finite Automaton

Now we come to the core part of the project, which is the implementation of *createFiniteAutomata* function, and is the required algorithm. Since it is a long function, it will be discussed in parts.

The first part is as follows

Text

Description automatically generated

Fig. 3

Here we take care of the invalid cases that can exists in the user input of the regular expression. If the user inputs an empty string, then there is no FA need to be created. Similarly, the ­*Kleene state operator \** cannot come at the front of an RE. In both cases, the program does not create an FA and returns.

If a valid regular expression (from the starting) is given, a start state will be created. It is now our current state on which machine is in right now for reading the RE. Let us take an example to go along with as we are creating the FA.

It is the language of all *(a, b)* such that there is exactly one double-*b* and any number of *a’s* in it. A start state is created for this RE.

The second part of the algorithm is the loop which traverses the whole RE and keeps making the DFA along the way. It is as follows

Text

Description automatically generated

Fig. 4

The *if* part in line 24 is the get rid of extra white spaces anywhere in the RE. The *else if* parts in lines 27 is for creating and *FA3* when *FA1* and *FA2*are given (used later, not here). It is like this

The input alphabet *(a, b)* will be detected at line 44. Inside it, it will put that character in the *lastRead* character variable, so that next state can be determined to know which state machine will go to after reading this letter in the next traversal of the loop. This will happen for the first character, which, in our example is an *a.*

Next time *\** is read at line 31. It means we must loop the *lastRead* character. Hence, the current state will put the address of its own state to one of the pointers *readA* or *readB* if the lastRead was an *a* or *b* respectively. After this, the state machine is as follows (based on the ongoing example)



Next time *b* is read as program reaches line 44. The *lastRead* is not a dash, so the code in the *inner-if* statement runs. It creates a new state. Then connects the current state to this new state by putting the address of the new state in current state’s *readA* or *readB* pointer depending on which letter is just read. Then the new state becomes the current state, and the currently read letter becomes the *lastRead.* Now the state machine looks like this

a

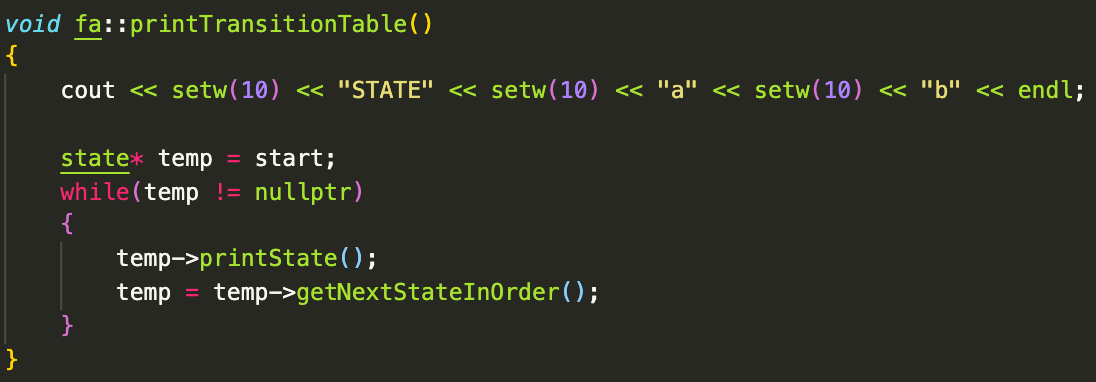
b

From the current state, the loop runs again, and it reaches line 44 and enters the *if* statement. Another state is created. This new state is then linked with the current state through *readA* pointer. Then the new state becomes current.

When a is read the same thing occurs. Same with the last *\**. The final *a* state is a final state so its *isFinal* will be set to *true.*

# Print Transition Table

When the FA is built, it is required to display it on the *console screen*. For that purpose, the *transition table* is shown. *Transition Table* completely represents an FA. Its code is as follows

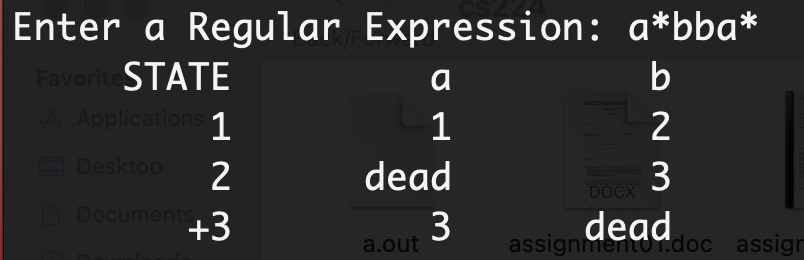


The structure of the transition table is as follows

|  |  |  |
| --- | --- | --- |
| STATE | a | b |
| 1 | destination state | destination state |
| 2 | destination state | destination state |
| +3 | destination state | destination state |

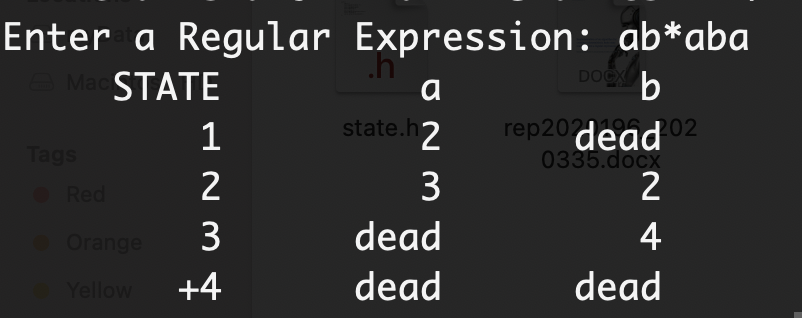
Where *destination state* is a member of *Q, finite set of states.* The *+* represents the final state. Customarily, the first state in the table is the start state.

For our ongoing example of the previous section, the transition table is given by



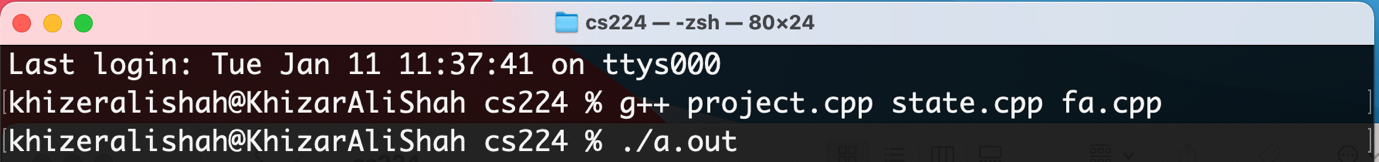
­­­

# Another example



# Compilation

The project consists of three *.cpp* files. Therefore, the compilation command is different from that of a single file. It is as follows



­­­g++ is the C++ compiler which compiles the given three files. Then after linking, an executable file is created. It is name *a.out.* Hence, running this file starts the program. It can be done by typing *./a.out* on a Macintosh computer, or simply *./a* or only *a* on a windows computer.

# Limitations:

* It will only work for an input alphabet of two symbols/letter, say 0 and 1, or ‘a’ and ‘b’.
* The use of brackets () and + make the algorithm very complex, hence they are kept away for the current project.